Local Storage Guidelines

Play+ Caching & Storage Helper Introduction ■ In the Play+ ecosystem, how we handle client-side data is as important as how we display it. This helper is based on the concept of Intentional and Resilient Storage , providing a safe and intelligent abstraction layer over the browser's raw localStorage API. Direct use of localStorage is discouraged as it's primitive, error-prone, and insecure for sensitive information. playcache is designed specifically for caching non-sensitive data like user preferences, UI state, and temporary application data. This aligns with our core design pillars by creating an Intuitive API that prevents common errors, an Adaptive system that can handle data expiration, and a Distinct , consistent pattern for storage across all projects. Package Info ■ The Play+ storage helper is included by default in the Golden Path starter kit. For existing projects, it can be installed via its dedicated package. Description ■ Package / Path Description Golden Path (Recommended) Pre-installed (/system/play.cache.ts) Uplift Path npm install @playplus/storage To get started with the Uplift Path, you can also run: npx playcache Folder Reference ■ File / Directory Purpose & Guidelines system/play.cache.ts The core storage service. It wraps the localStorage API, providing all helper methods. config/play.cache.config.json An optional file for overriding default caching behaviors, such as the key prefix. src/app/services/playcache.service.ts Angular service wrapper for reactive caching patterns. Helper - Pillars Alignment ■ Pillar How This Helper Aligns Intuitive Primary Pillar: Replaces the error-prone, manual process of using localStorage with a simple, safe, and memorable API. Adaptive Supports reactive data patterns through its event system, allowing the UI to adapt to storage changes across tabs. Distinct Enforces a single, consistent, and safe way to interact with browser storage across all Play+ applications. Helper Overview ■ The playcache helper is a smart and resilient wrapper around the browser's localStorage . Its purpose is to abstract the plumbing of client-side caching, so developers can store and retrieve non-sensitive data without worrying about common pitfalls. It automates and simplifies: Serialization : Automatically runs JSON.stringify() on set and JSON.parse() on get, preventing runtime errors and eliminating boilerplate. Key Scoping : Automatically prefixes all keys with a unique identifier to prevent collisions with other browser tabs, third-party scripts, or other apps on the same domain. Data Expiration (TTL) : Supports a "Time-to-Live" for cached items, treating expired data as null. Reactivity : Provides an event-based system (on / off) so your app can react to storage changes, even across tabs. Error Handling : Graceful handling of storage errors and quota exceeded Size Management : Built-in cache size

monitoring and cleanup Config Options ■ Optional global configuration can be provided in config/play.cache.config.json . Config Variable Default Value Description Recommended Value keyPrefix playcache The prefix added to all keys to prevent collisions. Keep default defaultTtl null Default Time-to-Live in seconds. null means no expiration. null maxSize 5242880 Maximum cache size in bytes (5MB). 5242880 cleanupInterval 300000 Cleanup interval in milliseconds (5 minutes). 300000 Helper Methods ■ Core Methods ■ Method Name Description Signature set Stores a serializable value in the cache, with optional TTL (in seconds). set(key: string, value: any, options?: { ttl: number }): void get Retrieves a value from the cache. Returns null if key doesn't exist or has expired. get<T>(key: string): T | null remove Deletes a specific item from the cache by its key. remove(key: string): void clear Removes all items from the cache that have the Play+ key prefix. clear(): void on Subscribes to changes for a specific key. Callback is triggered on updates (even across tabs). on(key: string, callback: (newValue: any) => void): void off Unsubscribes a callback for a specific key to prevent memory leaks. off(key: string, callback: (newValue: any) => void): void Utility Methods ■ Method Name Description Signature has Checks if a key exists and is not expired. has(key: string): boolean keys Gets all keys that match a pattern (supports wildcards). keys(pattern?: string): string[] getSize Gets the size of cached data in bytes. getSize(): number cleanup Cleans up expired items from the cache. cleanup(): void Basic Usage ■ import { playcache } from "../system/play.cache" ; // Store a value playcache . set ("user-preferences" , { theme : "dark" , language : "en" }) ; // Retrieve a value const preferences = playcache . get ("user-preferences") ; // Store with TTL (30 seconds) playcache . set ("temp-data" , { timestamp : Date . now ( ) } , { ttl : 30 }) ; // Listen for changes playcache . on ("user-preferences" , (newValue) => { console . log ("Preferences updated:" , newValue) ; }) ; React: A Custom Hook for Reactive Caching ■ // hooks/usePlayCache.ts import { useState , useEffect , useCallback } from "react" ; import { playcache } from "../system/play.cache" ; export function usePlayCache < T > (key : string , defaultValue : T) : [T , (newValue : T) => void] { const [value , setValue] = useState < T > (( ) => playcache . get < T > (key) ?? defaultValue) ; const updateValue = useCallback ((newValue : T) => { playcache . set (key , newValue) ; } , [key]) ; useEffect (( ) => { const handleUpdate = (newValue : T) => { setValue (newValue ?? defaultValue) ; } ; playcache . on (key , handleUpdate) ; return ( ) => playcache . off (key , handleUpdate) ; } , [key , defaultValue]) ; return [value , updateValue] ; } function ThemeSwitcher ( ) { const [theme , setTheme] = usePlayCache ("ui-theme" , "light") ; return (< button onClick = { ( ) => setTheme (theme === "light" ? "dark" : "light") } > Current theme: { theme } </ button >) ; } Angular: A Reactive Service ■ // core/services/playcache.service.ts import { Injectable } from

"@angular/core" ; import { playcache } from "@playplus/core" ; import { Observable , fromEvent , map , startWith } from "rxjs" ; @ Injectable ({ providedIn : "root" }) export class PlayCacheService { observe < T > (key : string , defaultValue : T) : Observable < T > { return fromEvent < StorageEvent > (window , "storage") . pipe (map (( ) => playcache . get < T > (key) ?? defaultValue) , startWith (playcache . get < T > (key) ?? defaultValue)) ; } set < T > (key : string , value : T , options ? : { ttl : number }) : void { playcache . set (key , value , options) ; } } Angular Service Usage ■ import { Injectable } from "@angular/core" ; import { Observable } from "rxjs" ; import { PlayCacheService } from "../services/playcache.service" ; @ Injectable ({ providedIn : "root" }) export class UserPreferencesService { constructor (private cacheService : PlayCacheService) { } // Reactive theme preference getTheme ( ) : Observable < "light" | "dark" > { return this . cacheService . createUserPreferences ("theme" , "light") ; } // Set theme preference setTheme (theme : "light" | "dark") : void { this . cacheService . set ("user-preferences:theme" , theme) ; } // Temporary data with TTL setTemporaryData (data : any , ttlSeconds : number) : void { this . cacheService . setTemporary ("temp-data" , data , ttlSeconds) ; } } Component Integration ■ import { Component , OnInit , OnDestroy } from "@angular/core" ; import { Subject , takeUntil } from "rxjs" ; import { PlayCacheService } from "../services/playcache.service" ; @ Component ({ selector : "app-theme-switcher" , template : ` ` , }) export class ThemeSwitcherComponent implements OnInit , OnDestroy { currentTheme : "light" | "dark" = "light" ; private destroy$ = new Subject < void > ( ) ; constructor (private cacheService : PlayCacheService) { } ngOnInit ( ) : void { // Reactive theme caching this . cacheService . createUserPreferences < "light" | "dark" > ("theme" , "light") . pipe (takeUntil (this . destroy$)) . subscribe ((theme) => { this . currentTheme = theme ; this . applyTheme (theme) ; }) ; } ngOnDestroy ( ) : void { this . destroy$. next ( ) ; this . destroy$. complete ( ) ; } toggleTheme ( ) : void { const newTheme = this . currentTheme === "light" ? "dark" : "light" ; this . cacheService . set ("user-preferences:theme" , newTheme) ; } private applyTheme (theme : "light" | "dark") : void { document . documentElement . setAttribute ("data-theme" , theme) ; } } Additional Info ■ Why We Created This Helper ■ Using window.localStorage directly introduces several issues: Manual JSON serialization/deserialization leads to bugs. No native TTL or expiration. Global key collisions. No reactivity within the same tab. Encourages storing sensitive data insecurely. playcache solves these by adding scoping, reactivity, TTLs, and a safe abstraction. Security: For Non-Sensitive Data Only ■ playcache does not encrypt data and is not safe for sensitive information. Use playguard for secure token and auth storage via HttpOnly cookies. Data Classification ■ Data Type Storage Method Example Sensitive

playguard (HttpOnly cookies) Auth tokens, user credentials User Preferences playcache Theme, language, UI settings Temporary Data playcache with TTL Search results, form data Static Reference playcache Dropdown options, configuration Caching Strategies ■ User Preferences ■ // Long-term storage for user preferences playcache . set ("user-preferences:theme" , "dark") ; playcache . set ("user-preferences:language" , "en") ; playcache . set ("user-preferences:notifications" , true) ; UI State ■ // Temporary UI state that persists during session playcache . set ("ui-state:sidebar-collapsed" , true) ; playcache . set ("ui-state:last-visited-page" , "/dashboard") ; Temporary Data ■ // Data with automatic expiration playcache . set ("temp:search-results" , results , { ttl : 300 }) ; // 5 minutes playcache . set ("temp:form-data" , formData , { ttl : 1800 }) ; // 30 minutes Cross-Tab Communication ■ // Listen for changes across tabs playcache . on ("user-preferences:theme" , (newTheme) => { // Update UI when theme changes in another tab document . documentElement . setAttribute ("data-theme" , newTheme) ; }) ; Best Practices ■ Do's ■ Cache static reference data (e.g., dropdowns, themes, user preferences) Use TTLs for temporary data to avoid stale state Clear cache intentionally (e.g., on logout) Use reactive patterns for UI state that needs to update Monitor cache size to prevent quota exceeded errors Clean up expired items periodically Don'ts ■ Never cache sensitive data (e.g., PII, tokens, passwords) Don't exceed 5MB browser storage limits Don't use localStorage directly - always use playcache Don't cache dynamic data that changes frequently Don't forget to unsubscribe from cache events Performance Considerations ■ Cache Size Management ■ // Monitor cache size const size = playcache . getSize ( ) ; if (size > 4000000) { // 4MB playcache . cleanup ( ) ; } // Clean up expired items periodically setInterval (( ) => { playcache . cleanup ( ) ; } , 300000) ; // Every 5 minutes Memory Leak Prevention ■ // Always unsubscribe from cache events ngOnDestroy ( ) : void { playcache . off ('user-preferences:theme' , this . handleThemeChange) ; } Testing ■ Unit Testing ■ import { PlayCache } from "../system/play.cache" ; describe ("PlayCache" , ( ) => { let cache : PlayCache ; beforeEach (( ) => { cache = new PlayCache ( ) ; localStorage . clear ( ) ; }) ; it ("should store and retrieve values" , ( ) => { cache . set ("test-key" , "test-value") ; expect (cache . get ("test-key")) . toBe ("test-value") ; }) ; it ("should handle TTL expiration" , ( ) => { cache . set ("expiring-key" , "value" , { ttl : 1 }) ; expect (cache . get ("expiring-key")) . toBe ("value") ; // Wait for expiration setTimeout (( ) => { expect (cache . get ("expiring-key")) . toBeNull ( ) ; } , 1100) ; }) ; }) ; Developer Checklist ■ Am I only storing non-sensitive data (like UI preferences or temporary reference data)? For data that can become stale, have I set an appropriate TTL? Am I using playcache instead of localStorage directly? Am I using on() and off() or reactive patterns if my UI needs to respond to changes?

Am I clearing or removing cache when appropriate (e.g., logout)? Am I monitoring cache size to prevent quota exceeded errors? Am I cleaning up expired items periodically? Am I unsubscribing from cache events to prevent memory leaks? Troubleshooting ■ Common Issues ■ Cache not persisting across sessions Check if the browser supports localStorage Verify the key prefix is correct Ensure the data is serializable Cross-tab updates not working Verify you're using the on() method correctly Check if the storage event is being handled Ensure the key prefix matches Cache size exceeded Implement cache cleanup Use TTL for temporary data Monitor cache size regularly Performance issues Avoid storing large objects Use TTL for temporary data Clean up expired items

```
import { playcache } from "../system/play.cache";

// Store a value
playcache.set("user-preferences", { theme: "dark", language: "en" });

// Retrieve a value
const preferences = playcache.get("user-preferences");

// Store with TTL (30 seconds)
playcache.set("temp-data", { timestamp: Date.now() }, { ttl: 30 });

// Listen for changes
playcache.on("user-preferences", (newValue) => {
  console.log("Preferences updated:", newValue);
});


---

// hooks/usePlayCache.ts
import { useState, useEffect, useCallback } from "react";
import { playcache } from "../system/play.cache";

export function usePlayCache<T>(
  key: string,
  defaultValue: T
): [T, (newValue: T) => void] {
  const [value, setValue] = useState<T>(
    () => playcache.get<T>(key) ?? defaultValue
  );

  const updateValue = useCallback(
    (newValue: T) => {
      playcache.set(key, newValue);
    },
    [key]
  );

  useEffect(() => {
```

```
    const handleUpdate = (newValue: T) => {
      setValue(newValue ?? defaultValue);
    };
    playcache.on(key, handleUpdate);
    return () => playcache.off(key, handleUpdate);
  }, [key, defaultValue]);

  return [value, updateValue];
}


---

function ThemeSwitcher() {
  const [theme, setTheme] = usePlayCache("ui-theme", "light");
  return (
    <button onClick={() => setTheme(theme === "light" ? "dark" : "light")}>
      Current theme: {theme}
    </button>
  );
}


---

// core/services/playcache.service.ts
import { Injectable } from "@angular/core";
import { playcache } from "@playplus/core";
import { Observable, fromEvent, map, startWith } from "rxjs";

@Injectable({ providedIn: "root" })
export class PlayCacheService {
  observe<T>(key: string, defaultValue: T): Observable<T> {
    return fromEvent<StorageEvent>(window, "storage").pipe(
      map(() => playcache.get<T>(key) ?? defaultValue),
      startWith(playcache.get<T>(key) ?? defaultValue)
    );
  }

  set<T>(key: string, value: T, options?: { ttl: number }): void {
    playcache.set(key, value, options);
  }
}


---

import { Injectable } from "@angular/core";
import { Observable } from "rxjs";
import { PlayCacheService } from "../services/playcache.service";

@Injectable({ providedIn: "root" })
export class UserPreferencesService {
```

```
  constructor(private cacheService: PlayCacheService) {}

  // Reactive theme preference
  getTheme(): Observable<"light" | "dark"> {
    return this.cacheService.createUserPreferences("theme", "light");
  }

  // Set theme preference
  setTheme(theme: "light" | "dark"): void {
    this.cacheService.set("user-preferences:theme", theme);
  }

  // Temporary data with TTL
  setTemporaryData(data: any, ttlSeconds: number): void {
    this.cacheService.setTemporary("temp-data", data, ttlSeconds);
  }
}


---

import { Component, OnInit, OnDestroy } from "@angular/core";
import { Subject, takeUntil } from "rxjs";
import { PlayCacheService } from "../services/playcache.service";

@Component({
  selector: "app-theme-switcher",
  template: `
    <button (click)="toggleTheme()">Current theme: {{ currentTheme }}</button>
  `,
})
export class ThemeSwitcherComponent implements OnInit, OnDestroy {
  currentTheme: "light" | "dark" = "light";
  private destroy$ = new Subject<void>();

  constructor(private cacheService: PlayCacheService) {}

  ngOnInit(): void {
    // Reactive theme caching
    this.cacheService
      .createUserPreferences<"light" | "dark">("theme", "light")
      .pipe(takeUntil(this.destroy$))
      .subscribe((theme) => {
        this.currentTheme = theme;
        this.applyTheme(theme);
      });
  }

  ngOnDestroy(): void {
    this.destroy$.next();
    this.destroy$.complete();
  }
```

```
  toggleTheme(): void {
    const newTheme = this.currentTheme === "light" ? "dark" : "light";
    this.cacheService.set("user-preferences:theme", newTheme);
  }

  private applyTheme(theme: "light" | "dark"): void {
    document.documentElement.setAttribute("data-theme", theme);
  }
}
```

```
// Long-term storage for user preferences
playcache.set("user-preferences:theme", "dark");
playcache.set("user-preferences:language", "en");
playcache.set("user-preferences:notifications", true);
```

```
// Temporary UI state that persists during session
playcache.set("ui-state:sidebar-collapsed", true);
playcache.set("ui-state:last-visited-page", "/dashboard");
```

```
// Data with automatic expiration
playcache.set("temp:search-results", results, { ttl: 300 }); // 5 minutes
playcache.set("temp:form-data", formData, { ttl: 1800 }); // 30 minutes
```

```
// Listen for changes across tabs
playcache.on("user-preferences:theme", (newTheme) => {
  // Update UI when theme changes in another tab
  document.documentElement.setAttribute("data-theme", newTheme);
});
```

```
// Monitor cache size
const size = playcache.getSize();
if (size > 4000000) {
  // 4MB
  playcache.cleanup();
}

// Clean up expired items periodically
setInterval(() => {
```

```
  playcache.cleanup();
}, 300000); // Every 5 minutes


---

// Always unsubscribe from cache events
ngOnDestroy(): void {
  playcache.off('user-preferences:theme', this.handleThemeChange);
}


---

import { PlayCache } from "../system/play.cache";

describe("PlayCache", () => {
  let cache: PlayCache;

  beforeEach(() => {
    cache = new PlayCache();
    localStorage.clear();
  });

  it("should store and retrieve values", () => {
    cache.set("test-key", "test-value");
    expect(cache.get("test-key")).toBe("test-value");
  });

  it("should handle TTL expiration", () => {
    cache.set("expiring-key", "value", { ttl: 1 });
    expect(cache.get("expiring-key")).toBe("value");

    // Wait for expiration
    setTimeout(() => {
      expect(cache.get("expiring-key")).toBeNull();
    }, 1100);
  });
});
```